TR08-0007

Final Report – Event Logic Assistant (Élan)

Mark Bickford

July 14, 2008

Prepared for:
Dr. Robert Herklotz
Air Force Office of Scientific Research
801 North Randolph Street
Room 732
Arlington, VA 22203-1977

Prepared by:
ATC-NY (Odyssey Research Associates)
Cornell Business & Technology Park
33 Thornwood Drive, Suite 500
Ithaca, NY 14850-1250

Contract No. FA9550-07-C-0150

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
14-07-2008	Final Report	15-09-2007 to 14-04-2008

4. TITLE AND SUBTITLE
Event Logic Assistant (Elan)

5a. CONTRACT NUMBER
FA9550-07-C-0150

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
0001AC

6. AUTHOR(S)
Dr. Mark Bickford

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
ATC-NY
33 Thornwood Drive, Suite 500, Ithaca NY 14850

8. PERFORMING ORGANIZATION REPORT NUMBER
TR08-0007

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Office of Scientific Research/NL
801 North Randolph Street
Room 732
Arlington, VA 22203-1977

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
AFRL-SR-AR-TR-08-0560

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release: Distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Report developed under STTR contract for topic AF07-T019. Distributed systems, now crucial to the infrastructure of our nation, are difficult to understand and design. The most effective way to gain intellectual control over complexity has always been abstraction. ATC-NY and Cornell University have developed the event logic formalism to permit development of distributed systems at a high level of abstraction. It provides an implementation-independent way to describe distributed computation and system requirements. This report describes the design of an Event Logic Assistant (Élan) that provides powerful automated support for applying event logic to the design and implementation of high-assurance distributed protocols. Its guiding principle is that the developer should perform all creative work•formalizing requirements and designing algorithms•at the highest possible level of abstraction, while automated tools take care of the rest.

15. SUBJECT TERMS
distributed systems, formal methods, event logic, automated proof

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Richard Smith
U	U	U	U	32	19b. TELEPHONE NUMBER *(Include area code)* 607-257-1975

Reset

Abstract

Distributed systems, now crucial to the infrastructure of our nation, are difficult to understand and design. The most effective way to gain intellectual control over complexity has always been abstraction. ATC-NY and Cornell University have developed the *event logic* formalism to permit development of distributed systems at a high level of abstraction. It provides an implementation-independent way to describe distributed computation and system requirements.

This report describes the design of an Event Logic Assistant (Élan) that provides powerful automated support for applying event logic to the design and implementation of high-assurance distributed protocols. Its guiding principle is that the developer should perform all creative work—formalizing requirements and designing algorithms—at the highest possible level of abstraction, while automated tools take care of the rest.

Élan factors the problem straightforwardly: capture a high-level description of requirements in event logic; refine those requirements to a distributed algorithm described in $E^{\#}$, a notation for "event logic programming"; generate code from $E^{\#}$. We present a detailed outline of $E^{\#}$ and its semantics, identify the principal issues in type checking and compilation, and describe enhancement we have made to the NuPrl theorem-proving environment to support interactive proofs that $E^{\#}$ programs meet their high-level specifications.

1 Introduction

Both military and commercial distributed systems require complex protocols that must behave as intended in all scenarios. Such systems have become too complex to understand, develop, and maintain without mathematical techniques and automated tools that support them. The most successful means for handling complexity has always been abstraction. In the past several years, ATC-NY and Cornell University have developed the *event logic* formalism to permit development of distributed systems at a very high level of abstraction. It provides an implementation-independent way to describe distributed computation and system requirements, and can be thought of as a formal semantic model of message sequence charts.

This report describes Phase I research on developing an Event Logic Assistant (Élan) that provides powerful automated support for applying event logic to the design and implementation of highly reliable distributed protocols. The initial design of Élan factored development into three steps:

1. Formalize the requirements in a high-level abstract model called *event logic* (section 2.1).

   Event logic specifications, a formalized version of message sequence diagrams, are easy to understand.

2. Refine the specifications to *abstract information flow constraints* (section 2.2).

   The constraints, also expressed in event logic, provide an abstract, machine-independent description of a distributed algorithm. NuPrl can be used, interactively, to prove that the constraints imply the formal high-level requirements. Section 3 describes enhancements to NuPrl in support of this.

   An abstract, machine-independent description of a distributed algorithm will allow us to provide interfaces to a variety of tools, such as model checkers and SAT-solvers, that can be used to verify properties of the algorithm.

   Previous work on event logic developed an interface to the ZChaff SAT-solver and used it to prove example lemmas about event logic. We have not pursued this connection in Élan Phase I, but Élan Phase II will provide interfaces to such tools.

3. Solve the constraints.

   Solving the information flow constraints amounts to deriving a program that implements them. One way to do so is to derive a *message automaton* that *realizes* the constraints and apply tools that we have already prototyped for generating code from an automaton. Message automata are described in section 2.1.2.

A key result of this research has been to refactor the problem, for reasons described in section 2.3. Instead of refining to information flow constraints and solving the constraints, we refine to "$E^{\#}$ programs"—which provide a highly structured way of describing information flow constraints. This structure provides a much clearer path for automating the process of solving constraints, which is equivalent to compiling $E^{\#}$.

At this stage, $E^{\#}$ is a proposal, not a language. Section 4 provides a detailed outline of $E^{\#}$ and section 5 identifies the principal issues in typechecking and compiling it.

Section 6 summarizes the results and briefly describes their significance as a basis for Phase II research.

2 Background

2.1 Event logic

2.1.1 Event structures

Intuitively, an event structure is an abstract algebraic construct that represents one possible execution history of a distributed system. An event logic *specification* of a system is a logical proposition about event structures: it asserts that all execution histories of the system satisfy the proposition.

Event structures can be thought of as a formal and general representation of *message sequence charts*, a common way in which developers represent distributed systems. The execution of a distributed system is characterized by a collection of *events*. Each event e occurs at a unique *location*, denoted $\widehat{e}$. A location is an abstraction of an agent or a process.

We can define these notions formally as follows. An *event language* is a multi-sorted language containing the following symbols:

- a sort E of events
- a sort Loc of locations
- a relation $<_c$ on events (the causal order relation).
- a function $\widehat{\ } : E \to Loc$

We call this set of symbols the *high-level core* of event logic. An event language may contain other symbols.

An *event structure* is an interpretation of an event language such that

1. The relation $<_c$ is a well-founded, transitive order on E, called *causal order*
2. Restricted to the events at any single location, $<_c$ is a total order.
   (We write $e' <_{loc} e$ when $\widehat{e'} = \widehat{e}$ and $e' <_c e$).

2.1.2 Message automata

Event structures are very abstract. We need a way to relate them to actual programs. In previous work we developed one way to do that, in terms of *message automata*.

We define an event language $\mathcal{E}$ that contains additional concepts such as: communication links, the sending and receiving of messages on named links, local state variables, and a way of labeling events with "kinds."

A message automaton is an abstract program expressed in these terms. Each message automaton consists of a finite number of clauses, such as

- when an event of kind k occurs, send the contents of local variable x on link l
- when an event of kind k occurs, set local variable x to `false`
- the only events that affect local variable x are events of kinds k, k'

We define the semantics of message automata[1] by defining a relation $\mathit{Consistent}(es, M)$ saying, intuitively, that event structure es is consistent with some execution of M. An automaton is *feasible* if at least one event structure is consistent with it (i.e., M is internally consistent).

Recall that an event logic specification ψ is a logical proposition about event structures. An automaton M *realizes* a specification ψ if M is feasible and if every event structure consistent with M satisfies ψ. Formally,

$$M \textbf{ realizes } \psi \;\Leftrightarrow\; \exists es.\, \mathit{Consistent}(es, M) \;\wedge\; \forall es.\, \mathit{Consistent}(es, M) \;\Rightarrow\; \psi^{es}$$

An event logic specification $\psi \in L$ is *realizable* (written $\textbf{realizable}(\psi)$) iff

$$\exists M.\, M \textbf{ realizes } \psi$$

A constructive proof of $\textbf{realizable}(\psi)$ provides a realizer; and in previous work we have implemented a translator that synthesizes Java code from message automata. Thus, synthesis of correct-by-construction code for specification ψ is reduced to proving $\textbf{realizable}(\psi)$. We have developed a number of tactics to help carry out these proofs interactively in NuPrl.

2.2 Information flow constraints

The logical picture described in sections 2.1 and 2.1.2 is unstructured. It provides no method for proceeding from the specification to a message automaton that realizes it. We developed a more disciplined method—factor the argument through an intermediate stage that abstracts the algorithmic core as *abstract information flow constraints*, leaving two verification tasks:

[1] The example shows instances of three of the six basic clauses used to defined message automata.

- Prove that the information flow constraints imply the original requirements.

  This step is carried out interactively in NuPrl. In Phase I we have improved the automated support for this step by adding *congruence closure* techniques to NuPrl (section 3).

- Solve the constraints.

  We showed how to compile information flow constraints that take the special form of *information flow on a graph*. We can compile constraints expressed in this special form by extracting a message automaton and applying tools developed in previous work to generate Java code from the automaton.

Our method for describing an algorithm abstractly is to define an *event class* that provides a view of the relevant system events and a characterization of certain causal relations among them.

2.2.1 Event classes

An *event class* consists of a set of events and a map that associates information (a value in some specified type) with each of them. If V is an event class we let

- $E(V)$ denote its associated set of events
- $V(e)$ denote the information associated with an event $e \in E(V)$

The interface to a component is an important example of an event class. Its events are the events occurring at the interface; typically, the associated information consists of the input and output values communicated by those events.

2.2.2 Antecedent functions

For any event class V, an *antecedent function* on V is a function $f : E(V) \rightarrow E(V)$ such that, for any event $e \in E(V)$, $f(e)$ is causally before or equal to e—i.e., $f(e) \leq_c e$.

Intuitively, the information available at event e must come either locally, from prior events at the same location as e, or externally from an event at some other location. In describing an algorithm abstractly, we use $f(e)$ (where f is an antecedent function) to indicate the source, if any, of external information available at e.

Given an antecedent function f, we'll say that

- e is *initial* if $f(e) = e$

  That is, the information at e does not come from elsewhere.

- e *propagates* if there's some $e' \neq$e such that $f(e') = e$

  That is, some other event e' gets information from e.

2.2.3 Information flow constraints

The information flow constraints for an abstract algorithm are expressed in terms of an event class V and an antecedent function, f, on V. The constraints typically have the following form:

- Initialization: If e is initial, then $P(\widehat{e}, V(e))$

  Property P, which relates locations to values, must hold for the location and value of every initial event.

- Propagation: If e is not initial, then $R(\widehat{e}, \widehat{f(e)}, V(e), V(f(e))$

  R relates the location and value of an event to the location and value of its antecedent.

- Termination: e propagates unless Q($\widehat{e}$,V(e))

  Information keeps on propagating until a certain stopping condition Q holds.

2.3 Lessons learned

The results of section 2.2 leave us with the problem of transforming—automatically, if possible—more general information flow constraints into information flows on a graph.

Our original hypothesis was that a collection of heuristic transformation techniques could be developed that would apply to typical information flow constraints to transform them into an information flow on a graph. We tested our hypothesis on the information flow constraints for a simple algorithm for *leader election in a ring*. Even for this relatively simple algorithm, we found that the generic methods that we had envisioned were not practical.

The problem was that the abstract constraints can refer to the entire past history of events, but an efficient algorithm must remember only a small part of that history or be able to accumulate a small "compression" of that history. We had hypothesized that we could automatically synthesize, from the logical form of the information flow constraints, an efficient compression function on the history.

Working on this hypothesis we developed an initial concept of a *programmable class* in which the value of an event in the class is a function of the full history of values of prior events in the class. We built a prototype compiler that attempted to optimize the function with which the class recognizes and assigns values to an event. The goal of this optimization was to replace the dependence on the full history by a dependence on something smaller. The method used in the prototype was based on recognizing certain functions of the full history of values that could be rewritten into equivalent functions expressed in terms of an accumulator. The algorithm then tried to combine the accumulators for the subexpressions into one global accumulator for the entire class.

The lesson learned from this exercise was that for the classes that occurred even in our relatively simple leader election algorithm, the method based on rewriting would not suffice. It did not succeed in finding an accumulator that efficiently compressed the state information needed by the program.

Analyzing the reasons for this failure, we saw that more structure would be required for event classes to be efficiently programmed. This analysis led to the development of $E^{\#}$, an abstract language for programming in event logic.

2.4 Overview of $E^{\#}$

$E^{\#}$ is a flexible, extensible notation for "programming" in event logic, from which it will be possible to generate efficient code.

2.4.1 $E^{\#}$ programs

An $E^{\#}$ program consists of two things: a collection of *programmable event classes* and a collection of *propagation rules* and *propagation constraints*.

Programmable event class An event class is *programmable* if local information suffices to determine whether an event belongs to the class and to compute its value.

Propagation rules and constraints A *propagation rule* specifies how events in certain event classes cause events in other classes. A *propagation constraint* specifies that events in some event class can *only* be caused by events in certain others.

Semantics As explained in section 2.1, we formalize an execution history of a distributed system as an event structure—a collection of events together with the causal relations between them. With each $E^{\#}$ program we associate a predicate that defines a set of event structures—all possible executions consistent with the requirements of the program. Thus, verification that an $E^{\#}$ program meets some high-level requirement will be purely an exercise in logic.

Language A few basic propagation rules and event classes, together with a few basic combinators for combining them, will generate a rich collection of programs. The straightforward logical semantics makes it easy to extend $E^{\#}$ by adding sophisticated constructs defined in terms of the basics.

2.4.2 Generating code from $E^{\#}$ programs

We can naturally associate an implicit "local state" with every event: it records the values associated with all causally prior events. Any $E^{\#}$ program will straightforwardly correspond to a collection of communicating state machines that maintain this natural local state—essentially, each machine transmits all the information it has and saves all the information it receives.

It will not be practical to implement such state machines as-is. A program with an efficient implementation must eventually "forget" the values of prior events. The structure of a programmable event class makes this forgetting explicit and allows a code generator to limit the amount of information needed to recognize and respond to new events.

3 Congruence closure

3.1 The basic algorithm

One of the fundamental forms of reasoning is equality reasoning—which proceeds by "substituting equals for equals." For example, from the assumptions

1. $a = b$
2. $b = c$
3. $f(a, c) = x$
4. $x < y$

it follows that $f(c, c) < y$: by (1) and (2), $a = c$; substituting that into (3) gives $f(c, c) = x$; and substituting that into (4) gives $f(c, c) < y$.

Two expressions built from function symbols and atoms (constants or variables) are *equivalent*, under given hypotheses, if they can be proven equal from those hypotheses by applying substitutions of equals for equals. A congruence closure algorithm builds a data structure that keeps track of those equivalence classes. Given the hypotheses in the example, the expressions $f(c, c)$ and x will be equivalent, so will belong to the same equivalence class. Once that data structure has been built, answering the question "are expressions t_1 and t_2 equivalent?" becomes extremely efficient.

Many congruence closure algorithms are known, but exploiting them in NuPrl presents two difficulties:

- To produce a complete NuPrl proof we cannot simply accept an assurance from some external oracle that two terms are equal but must actually construct a NuPrl proof that they are equal.

- The proofs produced by congruence closure algorithms are untyped, but NuPrl is a type system and equality of terms is meaningful only with respect to a type assignment.

The algorithm described in [39] generates a congruence closure data structure in such a way that one can extract for any pair of equivalent expressions a direct proof that they are equal. We adapted this algorithm by coding it into the Lisp core services of NuPrl and defining tactics that treat the proof produced by the congruence closure algorithm as a "proof plan." Tactics that use congruences apply type inference algorithms (already part of NuPrl tactics) to fill in the types in that plan. If this succeeds in creating a valid NuPrl proof, proof of the goal can be completed automatically.

3.2 Matching modulo a congruence

We have extended our tactics to include *matching modulo congruence*. Here is an example, showing how the conclusion $P(x, y)$ is automatically derived from the hypotheses

1. $\forall z.\ Q(z) \Rightarrow P(z, f(z))$
2. $y = f(x)$
3. $Q(x)$

The tactic tries to match the conclusion $P(x, y)$ against (appropriate subterms of) the hypotheses modulo the congruence. In this case, because $f(x)$ and y are equivalent, $P(x, y)$ matches the subterm $P(z, f(z))$ under the substitution of x for z. Using that match, and hypothesis (1), proving the conclusion reduces to proving $Q(x)$; but $Q(x)$ is hypothesis (3).

3.3 Heuristics

We have implemented a general method for specifying heuristics that add to the hypotheses equations that we know to be true. An *equational heuristic* consists of a pattern and a collection of equations. Whenever some subterm of the goal matches the pattern, a corresponding instance of each equation will be added to the hypotheses. Applying equational heuristics is logically sound, because any time one of the added equations is used in a proof, NuPrl will require that it be proven true.

Here are some simple but useful equational heuristics:

- The pattern is **if** x **then** y **else** z and the equations are (**if true then** y **else** z) $= y$ and (**if false then** y **else** z) $= z$
- The pattern is x & y and the equations are (**true** & y) $= y$ and (x & **true**) $= x$

Similar equational heuristics add knowledge about standard operations on pairs and lists. We also add domain specific knowledge about the logic of events, such as: the location of the sender of a message is the source of the link on which the message is sent.

Each added equation is tagged with a special token. When one of these equations is used, the proof tactic consults the token to determine how to prove it. (Recall that this is obligatory insurance against adding an unjustified hypothesis.) All these improvements have been added to the `Auto` tactic, which automates a large body of strategies for proving propositions about event logic.

3.4 Measurements

To obtain a rough measure of the value of these improvements we created a database of inferences from our large library of NuPrl proofs. These proofs, some with hundreds of individual inference steps, were created interactively by invoking tactics at each goal. A tactic entered by the expert user might have a form like this:

```
By lemma xxx THEN Auto
             THEN Instantiate 3 with [a;b;c]
             THEN Auto
             THEN Try (tac1)
             THEN Try (tac2)
```

A tactic of the form `A THEN B` applies the tactic `A` to the current goal and generates a list of subgoals and applies tactic `B` to each of these subgoals. To create our database we divided complex tactics into their constituent parts so that we had all of the subgoals for each individual tactic `A`, `B`, etc.

In our database, a *node* consists of a subgoal and an individual simple tactic. We call a subgoal that was completed by its tactic a *leaf node* and a subgoal and tactic that generated further subgoals an *interior node*. From 4206 complete NuPrl proofs we created a database of 203,455 nodes—75,919 interior nodes and 127,536 leaves.

We expected that most of these leaf nodes were already completed by the old `Auto` tactic, so we were particularly interested in how many of the interior nodes of our old proofs could now be completed automatically by an improved `Auto` tactic that used congruence closure. There were 6685 such cases, in some of which the new `Auto` replaced many steps of the original proof.

By that rough measure, the extended congruence closure algorithm has automated about 10% of what had been previously been done interactively. We believe that there is still room for improvement in this general purpose algorithm.

4 $E^{\#}$: Programming in event logic

This section outlines $E^{\#}$. An $E^{\#}$ program defines a system in terms of events and their causal relations: Execution consists of recognizing events and acting on them, causing other events to happen.

Accordingly, an $E^{\#}$ program consists of *event classes* and *propagation rules*. Intuitively, an event class recognizes some set of events and assigns information to each event it recognizes. A propagation rule stipulates that certain events cause other events and defines the information propagated. Reasoning about programs often requires *propagation constraints* expressing the assumption that certain events can *only* be caused by certain others. We formalize these ideas straightforwardly.

4.1 Event class

An event class of type T is a partial function from the domain E of events into T. For the present discussion, T is any type definable in the NuPrl environment. Some abbreviations:

- If e is an event and X an event class,

  $$X(e) = \bot$$

  means that e is not in the domain of X.

- Event e belongs to (is recognized by) X, written $e \in X$, if

  $$X(e) \neq \bot$$

- If e is an event and X an event class, we define $e \in X[v]$ to mean

  $$e \in X \ \& \ X(e) = v$$

  In words, e belongs to X with value v. (Note that an event may belong to different event classes and they may assign it different values.)

- We define $v =$ **most recent** A **before** $e \mid P(v)$ to mean

  $$\exists e' <_{loc} e.\ e' \in A \ \wedge \ v = A(e') \ \wedge \ P(v) \ \wedge \ \forall e'' \in A.\ (e' <_{loc} e'' <_{loc} e) \Rightarrow \neg P(A(e''))$$

  In words, there is an A event prior to e with a value satisfying P, and v is the value of the most recent such event. If the predicate P is always true, we omit it and write $v =$ **most recent** A **before** e.

4.2 Programmable event class

If an event class is *programmable*, local information will suffice to compute whether an event belongs to the class and, if it does, to compute its value. The programmable classes will be either base classes or built from base classes by applying certain combinators. $E^{\#}$ is extensible by defining new combinators in terms of the basics. For simplicity's sake, this section explains the idea informally, and omits discussion of types and type correctness.

Because an efficient program cannot store a complete history of all prior events, we limit the amount of local state information a programmable class may use to: the value of the most recent events in a fixed set of other classes, and the value of the most recent event in its own class. In theory, this is no limitation at all because we can define a programmable class that accumulates the history of all events that have causally preceded it. In practice, however, this limitation will make it possible to synthesize efficient code from typical $E^{\#}$ programs—because programmers will not make use of classes that accumulate unbounded amounts of information without designing mechanisms by which unneeded state information can be purged.

Base classes:

- **receive**(l, t)

  Recognizes the arrival of messages on link l with tag t—where *link* is an abstraction of "port" and *tag* an abstraction of "header." The value assigned to a receive event is the message received.

- **receive**$(*, t)$

  Recognizes the arrival of messages on any link with tag t.

- **internal**$(name)$

  Every event internal to a process has a label (a generalized, abstract program counter), and this class recognizes the internal events labeled by the given name. For technical reasons,[2] the value it assigns to any event is a randomly chosen integer. As will be seen, we can easily use the basic combinators to define a class of internal events that assigns a deterministic value to each of its events.

- *Var*

  We will allow $E^{\#}$ programs to contain variables that denote unspecified programmable classes. That will allow us to define program modules that are parameterized by some as yet unspecified input events.

[2]To simplify the description of protocols that rely on randomizing operations.

Abstract state variables:

We define an operator $'$ on event classes so that for any event class, A, the class A' recognizes any event for which a strictly prior A event has occurred and assigns it the value of the most recent such prior event. The specification of this combinator in event logic is

$$e \in A'[v] \Leftrightarrow v = \textbf{most recent } A \textbf{ before } e$$

We call classes of the form A' *abstract state variables* because A' remembers the value of the most recent A event.

Basic combinators:

Let A, B, C, D be programmable classes. Then the following classes are also programmable:

- A^τ

  Recognizes A events. To each $e \in A[v]$ assigns the pair $\langle v, t\rangle$ where t is the time at which e occurs.

- **let** $X = f(A, \ldots, B; C', \ldots, D'; X')$

  This is the basic definition scheme for programmable classes. It requires f to be a partial function and says that

  $$X(e) = \begin{cases} \perp & \text{if } e \notin A \ \& \ \ldots e \notin B \\ f(A(e), \ldots, B(e), C'(e), \ldots, D'(e), X'(e)) & \text{otherwise} \end{cases}$$

  The key points are that an event in X must belong to one of the classes $A, \ldots, B$; and that the criteria for membership in X and the value of an event in X may also depend on the values of some auxiliary state variables $(C', \ldots, D')$ and the value of the previous X event (hence the reference to X').

In terms of these basic combinators we can define many others, for example:

- $A + B$

  Recognizes events that are A events or B events and distinguishes three cases:

  - A but not B (in which case, it assigns the value assigned by A)
  - B but not A (in which case, it assigns the value assigned by B)
  - Both A and B (in which case, it assigns the pair of values assigned by A and B)

  This class can be defined by **let** $A + B = f(A, B)$ for the appropriate function f.

- $h \circ A$

  Recognizes those $e \in A[v]$ such that $v \in \text{domain}(h)$ and assigns the value $h(v)$.

  This class is defined by **let** $h \circ A = h(A)$.

- $A \mid b$, where b is a predicate

  Recognizes those $e \in A[v]$ such that v satisfies b; assigns the same value as A.

  This class is defined by **let** $A \mid b = f(A)$, where f is the partial identity function whose domain consists of those v that satisfy b.

- A **or** B

  If classes A and B have the same type T and are disjoint then A **or** B recognizes events that are A events or B and assigns the same value. This class is $h \circ (A + B)$ where $h : (T + T) \to T$ forgets which part of the disjoint union the value comes from.

- **accum**(h, x, A)

  Recognizes A events and assigns to each event $e \in A[v_0]$ the value $h(v_0, h(v_1, h(\ldots, x)\ldots)$ where $v_1, \ldots$ are the values of all prior A events.

  This class is defined by

$$\begin{array}{rcl} \textbf{let accum}(h, x, A) & = & f(A; (\textbf{accum}(h, x, A))') \quad \text{where} \\ f(a, \bot) & = & x \\ f(a, z) & = & h(a, z) \end{array}$$

- $A^{\#}$

  Recognizes A events. To each $e \in A(v)$ assigns the value $\langle n, v\rangle$, where n is the number of prior A events that have occurred at $\widehat{e}$.

  This class is defined by $A^{\#} = \textbf{accum}(h, \langle 0, \bot\rangle, A)$, where $h(v, \langle n, v'\rangle) = \langle n + 1, v\rangle$.

- $A; B$

  Recognizes any event $e \in B$ that is locally preceded by some A event. If $e \in B[b]$, then the value assigned to e is a pair $\langle a, b\rangle$ where $a =$ **most recent** A **before** e.

  This class is defined by

$$\begin{array}{rcl} \textbf{let } A; B & = & f(B, A; A') \quad \text{where} \\ f(\bot, -, -) & = & \bot \\ f(b, \bot, \bot) & = & \bot \\ f(b, a, -) & = & \langle a, b\rangle \\ f(b, \bot, a') & = & \langle a', b\rangle \end{array}$$

The basic classes and combinators (which we may extend using the definition scheme) are sufficient to program all event recognition and value computations.

4.3 Propagation rules

Besides recognizing events and computing values, a distributed algorithm must act on events by propagating information. This aspect of the algorithm is described in $E^{\#}$ as a set of *propagation rules*.

- **start**(*name*)

  This is the base case rule. It guarantees the creation of one event of class **internal**(name).

- $A \Rightarrow$ **receive**(l, t)

  This is the basic rule for message propagation. It says that every A event occurring at the source of link l results in a receive event on link l, having the same value and having tag t. If we want to transmit not the value of each A event but some function of that value, we say $h \circ A =>$ **receive**(l, t).

- $A \Rightarrow_{mcast}$ **receive**(t)

  This is the multicast propagation rule. It is well-formed only for classes A with value type of the form list$(Link) \times T$. If $e \in A[\langle lnks, v\rangle]$ it sends v on every link in $lnks$.

  More formally, there will be a receive of v with tag t at the destination end of each link in $lnks'$, where

  $$lnks' = \{l \in lnks | src(l) = \widehat{e}\}$$

- $A \Rightarrow$ **internal**(*name*)

  This is the time-delay propagation rule. It is well-formed only for classes A with a value type that can be interpreted as a time.

  It says that for each event $e \in A[t]$, an internal event labeled *name* will occur precisely t time units later.

4.4 Semantics of propagation rules

Each propagation rule has a specification in event logic. The basic propagation rule, multicast, and time-delay all have a similar form, asserting that:

> causally after every $e \in A[v]$-event there will exist an $e' \in B[v']$ such that $e <_c e'$ and v, v' are appropriately related.

For example, in the basic propagation rule, $A \Rightarrow$ **receive**(l, t), the base class **receive**(l, t) plays the role of "B" and equality is the "appropriate relation" between values. This propagation rule corresponds to *reliable* message transmission on the given link; so the generated code must implement it with a reliable transport mechanism such as TCP.

TCP connections also guarantee FIFO message delivery. Using the $<_{loc}$ ordering, this property is easily expressed in event logic. Our default assumption is that a link l implements a reliable FIFO channel. But $E^{\#}$ will also allow the programmer to declare that a link l is assumed to be only reliable (and not assumed FIFO), or, alternatively, that link l is neither reliable nor guarantees FIFO ordering of messages. Such declarations will allow the generated code to make use of a transport mechanism such as UDP that uses less bandwidth but provides weaker guarantees.

4.5 Propagation constraints

Note that the specification, given above, of $A \Rightarrow B$ says that there is a mapping from A-events *into* B-events. It says that every A-event results in a B-event, but it does not say that every B-event must be caused by an A-event.

To state such converse requirements, $E^{\#}$ will also include propagation constraints of the form

$$A_1, \ldots, A_n \Leftarrow B$$

This constraint, saying that every B event must be causally preceded by an event in one (or more) of the A_i, has the logical specification

$$\forall e \in B.\ \exists e'.\ e' <_c e\ \wedge\ (e' \in A_1\ \vee\ \ldots\ \vee\ e' \in A_n)$$

We may use $A \Leftrightarrow B$ as a shorthand for the propagation rule $A \Rightarrow B$ and the propagation constraint $A \Leftarrow B$.

4.6 The semantics of a program

As shown above, event classes, propagation rules, and propagation constraints correspond to predicates on event structures. The semantics of an $E^{\#}$ program is the conjunction of all of them.

4.7 Examples

Example event class: counter events The class that recognizes internal events labeled "count" and whose values are, successively, 0, 1, 2, ..., is definable as $\pi_1 \circ \mathbf{internal}(count)^{\#}$, where π_1 is the projection function that selects the first value of an ordered pair.

Example event class: deterministic internal events The class that recognizes internal events labeled "zero" and always returns value 0 is definable as $g \circ \mathbf{internal}(zero)$, where g is the constant function that returns 0 on all inputs.

Example: Perfect clock The $E^{\#}$ program consisting of the following two rules defines a perfect clock.

$$\textbf{start}(\mathit{clock})$$
$$(\lambda v.\ 1) \circ \textbf{internal}(\mathit{clock}) \Leftrightarrow \textbf{internal}(\mathit{clock})$$

The propagation rule is the time-delay rule, because the right-hand class consists of internal events. Each event has the value 1, which specifies that the next event will occur one time unit later.

Example: Dissemination We wish to design a system with two kinds of inputs. The first kind of input is configuration data. Inputs of this kind at location i will inform process i of a set of links to neighbors with which it should communicate. Each event in Nbr, the class of configuration inputs, either adds or removes a link.

For an appropriate function h, the current set of neighbors is computed by the event class

$$\text{Config} = \textbf{accum}(h, \phi, Nbr)$$

The second kind of input is sensor data, represented by event class $Sensor$. The dissemination program will send sensor data to all current neighbors and also forward sensor data it receives to its neighbors. In order to control this dissemination process, the original sensor data will be tagged with its location and each forwarding process will add its location to the set of tags. A process will not re-forward information tagged with its own location ($E^{\#}$ programs include an expression **here** that evaluates to the location of the evaluating process).

The dissemination program consists of the following two propagation rules

$$\begin{array}{rcl} \mathit{Config}; \mathit{Tag} & \Rightarrow_{mcast} & \textbf{receive}(\mathit{dissem}) \\ \mathit{Config}; \mathit{Fwd} & \Rightarrow_{mcast} & \textbf{receive}(\mathit{dissem}) \\ & \text{where} & \\ \mathit{Tag} & = & (\lambda v.\langle v, \{\textbf{here}\}\rangle) \circ \mathit{Sensor} \\ \mathit{Fwd} & = & (\lambda\langle v, s\rangle.\langle v, s \cup \{\textbf{here}\}\rangle) \circ (\textbf{receive}(*, \mathit{dissem}) \mid (\lambda p.\textbf{here} \notin \pi_2(p))) \end{array}$$

The first of these rules sends the sensor data, tagged with the location, on all links in the most recent configuration. The second rule forwards disseminated sensor data that is not tagged with the current location to all links in the most recent configuration after adding the current location to the set of tags.

The program will also contain the propagation constraint

$$\mathit{Tag}, \mathit{Fwd} \ \Leftarrow \textbf{receive}(*, \mathit{dissem})$$

And we could write the entire program as

$$\mathit{Config}; (\mathit{Tag}\ \textbf{or}\ \mathit{Fwd}) \Leftrightarrow_{mcast} \textbf{receive}(\mathit{dissem})$$

4.8 Defining extensions to $E^{\#}$

The event class combinators listed in section 4.2 all have simple definitions and simple logical specifications. The principles of $E^{\#}$ allow new combinators to be defined easily. When we define a new combinator we will also derive a logical specification for it, and anticipate defining many combinators that express useful distributed programming patterns.

Here is an example of such a pattern: it recognizes receipt of a message from a remote site that responds to an earlier (asynchronous) request; and the value assigned is a pair consisting of the request and response values. We can generalize this pattern to the task of recognizing events in a class B for which an earlier event in class A has a "matching" value. We could express this pattern with a new combinator, $A;_R B$, where the parameter R is a relation defining what we mean by a matching value:

$$e \in (A;_R B)[\langle a, b\rangle] \Leftrightarrow e \in B[b] \ \wedge \ a = \textbf{most recent } A \textbf{ before } e \mid R(a, b)$$

But, there is a problem with the above specification. As stated, the only way to realize it with a programmable class is to accumulate the values of *all* A-events. The reason is that, without some restriction on the relation R, we cannot be sure that the value of any A-event will not match some future B-event (and be the most recent match). Thus, we can never "forget" the value of any A-event, so we would have to allow the state to increase unboundedly.

If we change the logical specification so that $A;_R B$ recognizes B-events for which an earlier A-event has an R-matching value *which has not also matched an intervening B-event*, then we can realize the combinator efficiently. The state will now include only A-events that have not yet matched a B-event. This state could still grow unboundedly, but for typical applications it would not. (To guarantee that the states is bounded, we could make a variant that stores at most N unmatched A-events.)
If we define $R'(e, a, b)$ to be

$$R(a, b) \ \wedge \ \forall e' \in B.\ e' <_{loc} e \Rightarrow \neg R(a, B(e'))$$

then the new specification of $A;_R B$ is

$$e \in (A;_R B)[\langle a, b\rangle] \Leftrightarrow e \in B[b] \ \wedge \ a = \textbf{most recent } A \textbf{ before } e \mid R'(e, a, b)$$

since $R'(e, a, b)$ says that a is a match to b that has not been matched by an intervening B-event.

To implement this we define a programmable event class that accumulates the values of unmatched A-events.

$$\begin{aligned} \mathit{Unmatched}(A, B, R) &= \textbf{accum}(h, \textbf{nil}, B + A) \quad \text{where} \\ h(a, L) &= \textbf{append } (a, L) \\ h(b, L) &= L - \{a' \mid R(a', b)\} \end{aligned}$$

Then, using the definition scheme, we can define $A;_R B$ by

$$A;_R B = f(B, (\mathit{Unmatched}(A, B, R))')$$

for a suitable function f. The function f must be undefined unless there is an unmatched prior A-value that matches the B-value. So f is defined by:

$$\begin{aligned} f(b, \bot) &= \bot \\ f(b, U) &= \langle a, b\rangle, \text{ if } a = \text{last } a' \in U \; s.t. \; R(a', b) \\ f(b, U) &= \bot, \text{ otherwise} \end{aligned}$$

We must prove that the logical specifications of our definitions imply the desired logical specification for the new combinator. We anticipate building a tool a tool for defining and specifying new combinators that will produce the statement of a theorem of event logic that we can export to NuPrl in order to justify the new combinator.

Another example of a programming pattern that can be expressed as a combinator in $E^{\#}$ is the gathering of responses (or other events) until a quorum or other threshold is attained. By this we mean that after an initiating event in some class A, we accumulate responses in class B until some function of the accumulated responses crosses a given threshold. This pattern is used, for example, in fault-tolerant consensus protocols where, after multi-casting some message, a process gathers a quorum of responses. The quorum is recognized by including enough responses from different agents.

5 Compiling $E^{\#}$

This section discussions typechecking $E^{\#}$ programs and generating code from them.

5.1 Typechecking

Each event class has an associated type—the type of the values it assigns to the events it recognizes. Any reasonable type system can be used, and $E^{\#}$ can be parameterized by that choice. Checking the type correctness of an $E^{\#}$ program will reduce to checking the correct typing of certain expressions in the chosen type system.

So that we may use NuPrl for correctness proofs, we require that $E^{\#}$ programs be straightforwardly interpretable in NuPrl. Thus, the type system chosen should either be NuPrl itself or easily interpretable in NuPrl.[3] It could, for example, be a subset of the type system in some functional programming language such as $F^{\#}$, or a subset of another higher-type logic such as PVS.

[3]The disadvantage of using the full type system of NuPrl is that type checking is not decidable. The same is true of some other powerful type systems such as PVS.

That decision will be pragmatic.
Whatever choice is made, checking $E^{\#}$ programs for type correctness will require:

- implementing an algorithm that reduces type checking problems for $E^{\#}$ to type checking problems in the chosen type system for values;
- reinterpreting any error messages from that type checker in terms that the $E^{\#}$ programmer will understand.

5.2 Code generation

For any programmable class defined in $E^{\#}$, the computations that recognize events and assign values to them depend, ultimately, on recognizing finitely many kinds of basic events and storing finitely many values derived from these basic events. The propagation rules can be implemented by sending messages and scheduling internal events after timeouts. Thus, there is a compilation algorithm that reduces any $E^{\#}$ program to a set of these basic actions.

A code generator for $E^{\#}$ will have two key modules: one unwinds $E^{\#}$ definitions into their basic actions and the other implements an evaluator for terms in the system of value types used for $E^{\#}$. (Terms in that language will express tests used in conditional execution, the content of messages to be sent, etc.)

In the SCorES project [8] we prototyped a code generator that translates message automata into Java. It implements the basic communication primitive of message automata—reliable, FIFO communication on a named link—by sockets. So nodes send messages whenever they choose, and recipients not ready to process the messages they receive must queue them. In many applications it is desirable to institute a form of flow control whereby senders queue outgoing messages until recipients indicate that they are prepared to receive them. Compiler pragmas should give users control over such *global* optimization decisions we will provide a menu of such standard design options.

6 Results and Discussion

6.1 Phase I

In Phase I we have defined a systematic method for using event logic to develop distributed protocols at a high level of abstraction and the key elements of the tool suite, Élan, that support it. In particular, we have developed a detailed outline of $E^{\#}$, a language for "event logic programming." $E^{\#}$ programs are abstract, but can be directly compiled into code. Because $E^{\#}$ is abstract it is feasible, we believe, to verify formally that an $E^{\#}$ program meets high level requirements stated in event logic. We have also added significant support to the NuPrl theorem prover for carrying out those proofs (congruence closure techniques).

The most significant theoretical result of Phase I is our definition of the concept of programmable classes and the formal proof that programmable classes are closed under the combinator

$$\textbf{let } X = f(A, \ldots, B; C', \ldots, D'; X')$$

The proof of this theorem is constructive, so it allows us to automatically construct a "program" for class X from programs for classes $A, \ldots, B, C, \ldots, D$. A program in this sense is

- A list, $[i, j, \ldots]$, of locations at which events in X may occur.
- For each location i, a list $[k_1, k_2, \ldots]$ of kinds at location i that events in class X may have.
- A tuple $\langle s_1 : ty_1 \text{ initially } x_1, s_2 : ty_2 \text{ initially } x_2, \ldots \rangle$ of typed state variables with initial values.
- For each location i and each k in the list of kinds for that location, a list of functions that define how the next value of each state variable s_n is computed from the current values of the state variables and the value v of the event of kind k.
- For each location i and each k in the list of kinds for that location, a "test" function of the current state variables and the value v of the event, that returns a disjoint union of either the value assigned to the event by class X or else an indication that the event is not in class X.

We have built a prototype parser and compiler for $E^{\#}$ that uses this constructive compilation result. For example, the following $E^{\#}$ program

```
let ;(A,B) = f(B,A') where f(b,none)=none
                           f(b,a)=pair(a,b)
                           end
let l1 = [a b x1]
let l2 = [b c x1]
let k1 = rcv l1 fwd
let k2 = rcv l2 fwd
k1: int
k2: (int,int)
k1;k1 => k2
```

defines a simple version of the $(A;_R B)$ combinator discussed is section 4.8. It defines named links between locations `a`, `b`, and `c`, and kinds `k1` and `k2`, that are receives on these links. The program declares that the value of a message of kind `k1` is an integer and that messages

of kind k2 will be pairs of integers. Finally, it asserts the propagation rule that events of class k1;k1 will send their value to cause an event of kind k2.

To accomplish this propagation, we must build a program to recognize events of class k1;k1 and compute their value. The prototype compiler provides the following program for this class

```
at location "b":
 state is [s:int+top initially ff]
 [when kind = rcv(link "x1" from "a" to "b","x1")
  update state with:
  [s := inl val]
  test is case s of inl(a) => <a,val> | inr(a) => ff]
```

This program says that events of class k1;k1 can only occur at location "b" (the destination of link l1). Only one state variable, s, of type int + top is needed. It is initially set to ff, which is a value on the right side of the disjoint union (this is because ff, the "false" of Nuprl's boolean type, is defined by ff = inr Ax and Ax is in the "don't care" type top). The kind k1 has been replaced by its full description rcv(link "x1" from "a" to "b","x1"), and this is the only kind that an event in class k1;k1 may have.

The compiled program says that when an event of kind k1 occurs we update the state variable s to inl val, where val is the value of the event, thus "remembering" that value.

Finally, the test function, case s of inl(a) => <a,val> | inr(a) => ff, says that an event of kind k1 is an event in class k1;k1 if and only if the state variable s has the form inl(a) for some a, and if so, the value assigned to the event by class k1;k1 is the pair <a,val>. This is the pair of the remembered value and current value.

6.1.1 Example compilation

A more complete example $E^{\#}$ program is shown in figure 1. For this example, our prototype compiler produces the "basic" program shown in figure 2.

The basic program is divided into instructions at each location. These instructions first declare the kinds that occur at that location together with the type of their values. Then the state variables needed at that location are declared with their type and initial value.

Each kind of event is then listed with its affect on the state and messages it must propagate.

Note that in this example, at location "b", an event of kind k1 will always be in the basic class k1, and it will be in the defined class k1;k1 if a previous event of kind k1 has occurred. Thus the two propagation rules k1 => k2 and k1;k1 => k3 require that events of kind k1 send, depending on the state, either one or two messages, with appropriate tags. This behavior is evident in the sends listed for k1, which is rcv(link "x1" from "a" to "b","input").

```
let ;(A,B) = f(B,A') where f(b,none)=none
                           f(b,a)=pair(a,b)
                           end
let count(A) = c(A,self') where c(a,none) = 0
                                c(a,n) = n+1
                           end
let l1 = [a b x1]
let l2 = [b c x1]
let l3 = [c b x1]
let k1 = rcv l1 input
let k2 = rcv l2 fwd
let k3 = rcv l2 fwdd
let k4 = rcv l3 ack
k1: int
k2: int
k3: (int,int)
k4: int
k1 => k2
k1;k1 => k3
count(k2) => k4
```

Figure 1: Example $E^{\#}$ program.

```
at location "b":
 kinds are [rcv(link "x1" from "a" to "b","input"):int;
            rcv(link "x1" from "c" to "b","ack"):int]
 state is [s:int+top initially ff]
 [when kind = rcv(link "x1" from "a" to "b","input")
  update state with:
  [s := inl val]
  and
  [send [<val:int, "fwd">/
         case s of inl(a) => [<a, val>:(int,int), "fwdd"] |
                   inr(a) => []]
  on link "x1" from "b" to "c"]];
at location "c":
 kinds are [rcv(link "x1" from "b" to "c","fwd"):int;
            rcv(link "x1" from "b" to "c","fwdd"):(int,int)]
 state is [s1:int+top initially ff]
 [when kind = rcv(link "x1" from "b" to "c","fwd")
  update state with:
  [s1 := case s1 of inl(n) => inl (n+1) | inr(n) => inl 0]]
  and
  [send case s1 of inl(n) => [<n+1:int, "ack">] |
                   inr(a) => [<0:int, "ack">]
  on link "x1" from "c" to "b"]];
```

Figure 2: Basic program for example in figure 1

To complete the code generation for this $E^{\#}$ program we must generate code in the target language that implements these instructions. This will involve setting up event handlers for the kinds given by the program that invoke the appropriate update methods and send the listed message (or messages).

6.2 Phase II

Phase I research has provided a solid basis for creating a prototype of the Élan tool suite. Élan can be developed in stages, beginning with a front end and code generator for $E^{\#}$, viable as a stand-alone product. The most straightforward target for code generation would be a functional programming language, such as Microsoft's $F^{\#}$, and Microsoft Research has expressed interest in that possibility.

We will also use $E^{\#}$ in a project with Cornell University, funded by Air Force Research Lab, Rome, NY, that will develop verified fault-tolerant distributed consensus algorithms and generate correct-by-construction code from them. In addition to validating our methods, this work will help to develop additional support for formal verification.

References

[1] Uri Abraham. On interprocess communication and the implementation of multi-writer atomic registers. *Theoretical Computer Science*, 149:257–298, 1995.

[2] Uri Abraham. *Models for Concurrency*, volume 11 of *Algebra, Logic and Applications Series*. Gordon and Breach, 1999.

[3] Uri Abraham, Shlomi Dolev, Ted Herman, and Irit Koll. Self-stabilizing ℓ-exclusion. *Theoretical Computer Science*, 266:653–692, 2001.

[4] Myla Archer and Constance Heitmeyer. Mechanical verification of timed automata: A case study. Technical report, Naval Research Laboratory, Washington, DC 20375, May 19, 1997. A shorter version of this report was presented at RTAS '96, Boston, MA, June 10–13, 1996.

[5] T. Bell and S.K. Rajamani. The SLAM project: Debugging system software via static analysis. In POPL'02: *Proceedings of the ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages*, 2002.

[6] Andrew Barber, Philippa Gardner, Masahito Hasegawa, and Gordon D. Plotkin. From action calculi to linear logic. In Mogens Nielsen and Wolfgang Thomas, editors, *Computer Science Logic, 11th International Workshop, Annual Conference of the EACSL, Aarhus, Denmark, August 23-29, 1997, Selected Papers*, volume 1414 of *Lecture Notes in Computer Science*, pages 78–97. Springer, 1998.

[7] Mark Bickford and Robert L. Constable. A Logic of Events. Cornell University Technical Report 2003-1893, 2003.

[8] Mark Bickford and David Guaspari. A Programming Logic for Distributed Systems. ATC-NY Technical Report TR05-0007. Final report, contract no. FA9550-04-C-0106.

[9] K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[10] Michel Charpentier and K. Mani Chandy. Towards a compositional approach to the design and verification of distributed systems. In Jeannette Wing, Jim Woodcock, and J. Davies, editors, *FM99: The World Congress in Formal Methods in the Development of Computing Systems*, volume 1708 of *Lecture Notes in Computer Science*, pages 570–589, 1999.

[11] Edmund M. Clarke and E. Allen Emerson. Synthesis of synchronization skeletons from branching time temporal logic. In *Proc. Workshop on Logics of Programs*, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer–Verlag, 1982.

[12] E. Allen Emerson and Edmund M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3):241–266, 1982.

[13] Kai Engelhardt, Ron van der Meyden, and Yoram Moses. A program refinement framework supporting reasoning about knowledge and time. In Jerzy Tiuryn, editor, *Proc. Foundations of Software Science and Computation Structures (FOSSACS 2000)*, pages 114–129, Berlin/New York, 1998. Springer-Verlag.

[14] Kai Engelhardt, Ron van der Meyden, and Yoram Moses. A refinement theory that supports reasoning about knowledge and time for synchronous agents. In Robert Nieuwenhuis and Andrei Voronkov, editors, *8th International Conference on Logic for Programming, Artificial Intelligence and Reasoning*, volume 2250 of *Lecture Notes in Artificial Intelligence*, pages 125–141. Springer-Verlag, December 2001.

[15] Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. *Reasoning About Knowledge*. Massachusetts Institute of Technology, 1995.

[16] Ronald Fagin, Joseph Y. Halpern, Yoram Moses, and Moshe Y. Vardi. Knowledge-based programs. *Distributed Computing*, 10(4):199–225, 1997.

[17] Joseph Y. Halpern. A note on knowledge-based programs and specifications. *Distributed Computing*, 13(3):145–153, 2000.

[18] Joseph Y. Halpern and Ronald Fagin. Modeling knowledge and action in distributed systems. *Distributed Computing*, 3(4):159–177, 1989.

[19] Joseph Y. Halpern and Richard A. Shore. Reasoning about common knowledge with infinitely many agents. In *Proceedings of the 14th IEEE Symposium on Logic in Computer Science*, pages 384–393, 1999.

[20] T.A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with BALST. *Proceedings of the 10th SPIN Workshop on Model Checking Software*, 2003.

[21] D. Harel and R. Marelly. *Come, Let's Play: Scenario-Based Programming Using LSCs and the Play-Engine*. Springer-Verlag, New York, 2003.

[22] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[23] Isabelle home page. `http://www.cl.cam.ac.uk/Research/HVG/Isabelle`.

[24] J. Klose and H. Wittke. An automata based interpretation of live sequence charts. In *Proceedings of Seventh International Coference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'01)*, 2001.

[25] K. Koskimies and E. Makinen. Automatic synthesis of state machines from trace diagrams. *Software–Practice and Experience*, 24(7):643–658, 1994.

[26] S. S. Kulkarni, J. Rushby, and N. Shankar. A case-study in component-based mechanical verification of fault-tolerant programs. In A. Arora, editor, *Proceedings of the 19th IEEE International Conference on Distributed Computing Systems Workshop on Self-Stabilizing Systems, Austin, TX*, pages 33–40. IEEE Computer Society Press, 1999.

[27] Leslie Lamport. Time, clocks and the ordering of events in a distributed system. *Comms. ACM*, 21(7):558–65, 1978.

[28] Leslie Lamport. Hybrid systems in TLA+. In Grossman, Nerode, Ravn, and Rischel, editors, *Hybrid Systems*, volume 736 of *Lecture Notes in Computer Science*, 1993.

[29] Leslie Lamport. The temporal logic of actions. ACM Transactions on Programming Languages and Systems (TOPLAS) 16(3):872–923, 1994.

[30] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, Boston, 2003.

[31] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, San Mateo, CA, 1996.

[32] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, Berlin, 1992.

[33] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer-Verlag, Berlin, 1995.

[34] Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. *ACM Trans. Program. Lang. and Syst.*, 6(1):68–93, 1984.

[35] R. Milner. Action structures and the π-calculus. In Helmut Schwichtenberg, editor, *Proof and Computation*, volume 139 of *NATO Advanced Study Institute, International Summer School held in Marktoberdorf, Germany, July 20–August 1, 1993, NATO Series F*, pages 219–280. Springer, Berlin, 1994.

[36] Robin Milner. *Communication and Concurrency*. Prentice-Hall, London, 1989.

[37] Robin Milner. Calculi for interaction. *Acta Informatica*, 33(8):707–737, 1996.

[38] Jayadev Misra. *A Discipline of Multiprogramming*. 2001.

[39] Robert Nieuwenhuis and Albert Oliveras. Proof Producing Congruence Closure. In J. Giesl, editor, *Proceedings of the 16th International conference on term rewriting and applications, Nara, Japan, 2005*, volume 3467 of *Lecture Notes in Computer Science*, pages 453–468. Springer, 2005.

[40] Lawrence C. Paulson. Mechanizing UNITY in isabelle. *ACM Transactions on Computational Logic*, 1999.

[41] A. Pnueli and R. Rosner. Distributed reactive systems are hard to synthesize. In *Proceedings of Thirty-first IEEE Symposium on Foundations of Computer Science*, pages 746–757, 1990.

[42] Shaz Qadeer and Natarajan Shankar. Verifying a self-stabilizing mutual exclusion algorithm. In David Gries and Willem-Paul de Roever, editors, *IFIP International Conference on Programming Concepts and Methods: PROCOMET'98*, pages 424–443, Shelter Island, NY, June 1998. Chapman & Hall.

[43] Fred B. Schneider. *On Concurrent Programming*. Springer-Verlag, New York, 1997.

[44] Gerard J. Holzmann *The Spin model checker: primer and reference manual*. Addison-Wesley, 2004.

[45] R. van Renesse and F. Schneider. Chain Replication for Supporting High Throughput and Availability *Proc. of the 6th Symposium on Operationg Systems Design and Implementation*, December 2004.

[46] M. Y. Vardi. An automata-theoretic approach to fair realizability and synthesis. In P. Wolper, editor, *Computer Aided Verification, Proceedings of the 7th International Conference*, volume 939 of *Lecture Notes in Computer Science*, pages 267–292. Springer-Verlag, 1995.

[47] G. Winskel. *Events in Computation*. PhD thesis, University of Edinburgh, 1980.

[48] G. Winskel. An introduction to event structures. In J. W. de Bakker et al., editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, number 345 in Lecture Notes in Computer Science, pages 364–397. Springer, 1989.

[49] Job Zwiers, Willem P. de Roever, and Peter van Emde Boas. Compositionality and concurrent networks: Soundness and completeness of a proofsystem. In *ICALP 1985*, pages 509–519, 1985.